Basics

- The differences between **OOP** and **OBP** language are:

Object Oriented Programming	Object Based Programming
It **follows** the concept of OOP.	It **doesn't** follow the concept of OOP.
It **doesn't** have inbuilt objects.	It **has** the inbuilt objects.

- If we **don't** add the **static** modifier in the main method signature, the compilation of the program will go through without any issues, but while executing it, a **NoSuchMethodError** will be thrown.
- ***Definition*:** A friend function is a function that is defined outside the class, but has the **right** to access all **private** and **protected** data members of the class.
- ***Definition*:** A syntax is defined as the set of rules, according to which, a Java program is written and interpreted.
- The differences between **POP** and **OOP** are:

Procedure Oriented Programming	Object Oriented Programming
Hear the program is divided into smaller parts known as **procedures** of **functions**.	Here the program is divided into smaller parts known as **objects**.
The programming language which follows this concept is called Procedure Oriented Programming Language (**POPL**).	The programming language which follows this concept is called Object Oriented Programming Language (**OOPL**).
Ex: C, COBOL, FORTRAN, etc.	Ex: C++, C#, Java, etc.

- ***Definition*:** In Java, function prototyping is defined as a declaration of a function that specifies the function's name and type signature with data types and parameters, but omits the functions body or definition. Ex: int add(int a);

Inheritance

- ***Definition*:** Inheritance is defined as the process in which objects of one class acquire the properties of objects of another class.
- Subclass refers to its **immediate** superclass by using **super** keyword.
- ***Uses*** of **super** keyword:
    - *To call superclass constructors*: **super(parameter-list);**
    - *To overcome name hiding*: **super.member;**

- In both of the above uses, the **super** keyword must always be the **first statement** to be executed inside a subclass constructor.
- ***Definition*:** A constructor in Java is a special method used for initializing the objects. The constructor is called when an object of a class is created.
- Constructor overriding is **never possible** in Java.
- The differences between a **Method** and a **Constructor** are:

Method	Constructor
Methods **should** have a return type.	Constructors **don't** have any return type.
They are used for **exposing** the **behavior** of an object.	They are used for **initializing** the **state** of an object.
These can be executed **multiple times** on an object.	These will get executed **only once** per object.
They are **inherited** by the subclass.	They are **not inherited** by the subclass.
Invoked **implicitly** by the **system**.	Invoked **explicitly** by the **programmer**.

- ***Definition*:** Constructor Chaining is the process in which one constructor calls another constructor w.r.t the current object. It occurs through inheritance. In Java, it's achieved in 2 ways:
  - Within same class
    - It can be done using **this()** keyword to call constructors of the same class.
  - From base class
    - It can be done using the **super()** keyword to call constructors of the base class.

Polymorphism

- Polymorphism is implemented either using **late binding** or **dynamic binding** (**runtime binding**).
- It's implemented in **two** ways:

Overloading	Overriding
It occurs when methods have the **same** method name with **different** lists of arguments and parameters.	It occurs when methods have the **same** method name with the **same** list of arguments and parameters.

Occurs within the **same class**.	Occurs within <u>both</u> **subclass** as well as **superclass**.
It's the example of **compile time** polymorphism	It's the example of **runtime** polymorphism

- A subclass can implement its own behavior using method **overriding**.
- A subclass includes all of the members (**default, public, protected**) of its superclass except **private** members.

N.B.

• **Superclass ref= new Superclass();**
  ○ This type of reference variable only has knowledge of the methods declared by that particular superclass itself.
• **Superclass ref= new Subclass();**
  ○ This type of reference variable only has knowledge of the methods declared by the superclass and not by the subclass.
• **Subclass ref= new Subclass();**
  ○ This type of reference variable only has knowledge of the methods declared by that particular subclass itself.

Dynamic Method Dispatch

- ***Definition*:** It's a mechanism in which a call to an **overridden** method is resolved at **runtime**, rather than compile time.
- When an overridden method is called through a superclass reference, the method to be executed will be based upon the type of the **object** being referred to at the time the call occurs & not the type of the **reference variable**.

Abstract Class & Methods

- ***Definition*:** A method that has been declared but not defined is an abstract method.
- A class that contains **one or more** abstract methods must also be declared abstract.
- An abstract class **can't** be instantiated, but a reference to an abstract class **can** be created.
- A class can be declared abstract even if it doesn't contain any abstract methods. This **prevents** the class from being **instantiated**.
- An abstract class **can** also have **concrete methods**.

- An abstract class **can** be **inherited**.
    - If the subclass **defines** all the inherited abstract methods, then it's called **complete** and hence can be instantiated.
    - If the subclass **doesn't define** all the inherited abstract methods, then it's called **incomplete** and hence must be declared abstract.

"Final" Keyword

- ***Uses*** of **final** keyword:

    - **To create a constant variable.**
        - Final variables must be initialised at the time of declaration.
    - **To prevent overriding**.
        - Final methods can't be overridden.
    - **To prevent inheritance.**
        - Final classes can't be inherited.

- Java resolves calls to **normal methods** at **run time** which is called **late binding**.
- Java resolves calls to **final methods** at **compile time** which is called **early binding**.

Object Class

- Object class is a **special** class defined by Java and is a **superclass of all other classes**.

Interface

- ***Definition*:** Interface is an abstract type that defines a standard and public way of specifying the behavior of a class.
- Interface specifies what a class must do, and not how it does it.
- A class can implement **any number** of interfaces.
- The **methods** & **variables** that are **declared** in an interface are **abstract** & **final** respectively **by default**.
- Variables can be **defined** in an interface, but **implicitly** they are final variables.
- A **concrete** class **must** implement an interface in order to define all the **abstract** methods of that interface.
- A class must be declared **abstract** if it **partially** implements the interface methods.
- An interface is always **preceded** either by **public** or by **default** access specifier.

- All the methods and variables declared within an interface are implicitly **public** if the interface itself is declared as **public**. Same will be the case for **default** access specifiers.
- The methods implementing an interface inside a class must be **preceded** with the keyword **public**.
- An interface **can't** be instantiated, but a reference to an interface **can** be created.

N.B.
• **Interface ref= new Class();**   ◦ This type of reference variable only has knowledge of the methods declared by the interface and not by the class implementing the interface.

- ***Uses*** of Interface:
  - To reveal an object's programming interface.
  - To have unrelated classes.
  - To model multiple inheritance.

Package

- ***Definition*:** A package represents a directory that contains a related group of classes and interfaces.
- In a Java source file, an **import** statement occurs **immediately** following the **package** statement, if it exists, and before any **class** definition.
- There is **no practical limit** on the depth of a package hierarchy, except that is imposed by the file system.
- All of the standard Java classes included with Java are stored in a package called **java**.
- **java.lang** package is **implicitly** imported by the compiler for **all** programs.
- Class Member Access: (***Privacy*** *is the* ***Default*** *of* ***Protected*** ***Public***)

		Private	Default	Protected	Public
Same class		Yes	Yes	Yes	Yes
Same Package	Subclass	No	Yes	Yes	Yes
	Non-subclass	No	Yes	Yes	Yes
Different Package	Subclass	No	No	Yes	Yes

	Non-subclass	No	No	No	Yes

N.B.
• The **public** members can be accessed everywhere. • The **private** members can be accessed only inside the same class. • The **protected** members are accessible to every child class (same or other package). • The **default** members are accessible within the same package but not outside the package.

Exception

- ***Definition*:** An exception is an abnormal condition or situation that arises in a code sequence at runtime.
- An **exception** is also called a **runtime error**.
- When an exceptional condition arises, an object representing that exception is created and is thrown into the method that caused the error.
- Java exception handling is managed through **5** keywords:
  - The **try** statement allows us to define a block of code to be **tested for errors** while it's being executed.
  - The **catch** statement allows us to define a block of code to be executed **if an error occurs** in the try block.
  - The **throw** statement allows us to **create a custom error**.
  - The **throws** clause **indicates** the type of exceptions that a **method** might throw.
  - The **finally** statement allows us to execute code after **try...catch**, **regardless of the result**.
- Java's unchecked RuntimeException subclass is defined in the package **java.lang**.

Checked & Unchecked Exception

- The differences between **checked** and **unchecked** exception are:

Checked Exception	Unchecked Exception
These are also called **compile-time** exceptions.	These are also called **runtime** exceptions.
These are propagated using **throws**	These are propagated **automatically**.

keyword.	

- List of some **unchecked exceptions** are:
  - ArithmeticException
  - NegativeArraySizeException
  - NullPointerException
  - ArrayIndexOutOfBoundsException
- List of some **checked exceptions** are:
  - ClassNotFoundException
  - IllegalAccessException
  - InterruptedException
  - InstantiationException

Chained Exception

- ***Definition*:** Chained Exception is a feature, which helps us to identify a situation where one exception creates another exception.
- The **initCause()** method **sets** the actual cause of an exception.
- The **getCause()** method **returns** the actual cause of an exception.
- The **initCause()** method can only be **accessed** using the **getCause()** method.
- If the **getCause()** method is called without the implementation of **initCause()** method, then the Java compiler will display **null**.

Static & Non-Static Methods

Static Method	Non-Static Method

class Main
{
static void demo()
{
System.out.print("Hello");
}
public static void main(String args[])
{
demo();
}
}
	class Main
	{
	static void demo()
	{
	System.out.print("Hello");
	}
	void demo1()
	{
	System.out.print(" World");
	}
	public static void main(String args[])
	{
	Main ref=new Main();
	ref.demo();
	ref.demo1();
	}
	}

Output: Hello	**Output:** Hello World

- The restrictions that are applied to the Java static methods are:
    - The static method **can't** use non-static data members or invoke non-static methods directly.
    - The **this** and **super** keywords **can't** be used in static members.
    - The static method can access **only** static type data.
    - A static method **can't** be **overridden** in a subclass.

Multi-Threading

- ***Definition*:** In Java, multi-threading is a feature that allows concurrent execution of 2 or more parts of a program for maximum utilisation of CPU. Each part of such a program is called thread.
- Multi-threading is a **specialized form** of **multitasking**.
- Multitasking is of **2** types:
    - **Process based** multitasking is the feature that allows our computer to run multiple programs simultaneously.
    - **Thread based** multitasking is the feature that allows our computer programs to run multiple sub-processes or tasks simultaneously.
- ***Definition*:** A **program** is the smallest unit of code that can be dispatched by the scheduler.
- ***Definition*:** A **thread** is the smallest unit of dispatchable code or a lightweight sub-process.
- The differences between **process** and **thread** are:

Process	Thread
These are **heavy-weight** tasks and require their **own** separate address space.	These are **light-weight** tasks and cooperatively **share** the same address space.
Inter-process communication is **expensive** and **limited**.	Inter-process communication is **inexpensive**.
Context switching from one process to another is **costly**.	Context switching from one thread to another is **cheap**.

- ***Definition*:** When **2 or more** threads need access to a **shared** resource, then they somehow need to figure out a way in order to ensure that the resource must be used by only **1 thread at a time**. So the process by which this is achieved is called **Synchronisation**.
- ***Definition*:** A **monitor** is an object that is used as a mutual exclusive lock or mutex.
- Only **one** thread can own a monitor at any given time.

- ***Definition*:** When 2 threads have a circular dependency on a pair of synchronised objects, then such a situation is called **Deadlock**.
- ***Definition*:** If 1 thread tries to **read** the data and another thread tries to **update** the **same** data at the **same** exact time, then it's known as the **read-write problem**.
- Read-write problems can be prevented by synchronising access to the data by using the keyword **synchronized**.
- **Producer-consumer** problem is also known as the **bounded-buffer** problem.
- Java supports **inter-thread communication** through **two** given methods, i.e. **wait()** and **notify()**.

Thread

- Threads can be created using **two** different mechanisms:
  - By **extending** the **Thread** class.
  - By **implementing** the **Runnable** interface.

Program

```
class A extends Thread
{
        public void run()
        {
                System.out.println("extends Thread");
        }
}
class B implements Runnable
{
        public void run()
        {
                System.out.println("implements Runnable");
        }
}
class Main
{
        public static void main(String args[])
        {
                A t=new A();
                t.start();

                Thread r=new Thread(new B());
                r.start();
        }
}
```

Output (Most Probable)	Output (Least Probable)
extends Thread implements Runnable	implements Runnable extends Thread

- The differences between **Thread** class and **Runnable** interface are:

Thread class	Runnable interface
Extending the Thread class **gives** developers the **full control** over the thread life cycle.	Implementing the Runnable interface **doesn't give** developers **any control** over the thread life cycle.
Extending the thread class **doesn't allow** the derived class to extend any other base classes.	Implementing the runnable interface **allows** the derived class to extend other base classes.

- **Thread scheduler** moves the thread from **Ready** state to the **Runnable** state.
- In Java, all the **developer created** thread instances have the **same** priority.
- The life cycle of a thread is:
    - A thread that's just instantiated is in a **New** state.
    - A thread executing in the JVM is in **Runnable** state.
    - A thread waiting for a **monitor lock** is in a **Blocked** state.
    - A thread that's waiting **indefinitely** for another thread to perform a particular task is in a **Waiting** state.
    - A thread that's waiting for another thread **up to a specified breaking time** in order to perform a particular task is in a **Sleeping** state.
    - A thread that has completed its task is in **Terminated** state.
- The **isAlive()** method returns **true** only if the thread upon which it's called is **still running**. Otherwise, it returns **false**.
- The **join()** method **prevents** the next thread from **executing** until the current thread, upon which the join() method is called **completes** its **execution**.

Nested Class

- ***Definition*:** A class written within another class is called **Nested Class**, i.e. an **Inner class** is called Nested Class.
- An Inner class can be declared with all access specifiers (**public, private, default, protected**), whereas an Outer class can only be declared either with the **public** or **default** access specifier.
- Types of Nested Class:

• **Static** nested class.

- **Non-static** nested class.
  - **Member** inner class.
  - **Method-Local** inner class.
  - **Anonymous** inner class.

- A static nested class **doesn't** have access to the **instance variables** and **methods** of the outer class.

Program

```
class Outer
{
	static class Inner
	{
		public void meth()
		{
			System.out.print("Hello");
		}
	}
	public void meth1()
	{
		System.out.print(" World");
	}
	public static void main(String args[])
	{
		Inner r=new Inner();
		r.meth();

		Outer r1=new Outer();
		r1.meth1();
	}
}
```

Output: Hello World

- Inner classes **can** be used as the **security mechanism** in Java.
- Once an inner class is declared as **private**, it **can't** be accessed **directly** through an object **outside** the class.
- Since the scope of a **method local inner class** is restricted within the method in which it's defined, hence such classes can be instantiated only within those methods only.
- An inner class declared **without** a class name is an **anonymous inner class**. Such classes can be **created** and **instantiated** at the **same** time.

- Anonymous inner classes are generally used whenever we need to **override** the **methods** of a class or an interface.

String

- ***Definition*:** A string is a sequence of characters which are generally **immutable** once it's created.
- **StringBuffer** classes are generally used for performing **changes** in original strings.
- **String** and **StringBuffer** classes are pre-declared **final**.
- The differences between **StringBuffer** and **StringBuilder** are:

StringBuffer	StringBuilder
It's **synchronized**, i.e. it **is** thread **safe**.	It **isn't synchronized**, i.e. it **isn't** thread **safe**.
Here, two threads **can't** call the methods of the StringBuffer **simultaneously**.	Here, two threads **can** call the methods of the StringBuilder **simultaneously**.
It's **less efficient** than StringBuilder.	It's **more efficient** than StringBuffer.

String Tokenizer

- The **StringTokenizer** class allows an application to break a string into tokens.
- The **StringTokenizer** methods don't distinguish among identifiers, numbers, and quoted strings, nor do they recognize and skip comments.

Wrapper Class

- ***Definition*:** The Wrapper class in Java provides the mechanism to convert **primitive** data type into **object** and **vice versa**.
- ***Definition*:** The automatic conversion of primitive data type into its corresponding wrapper types is known as **autoboxing**.
- ***Definition*:** The automatic conversion of wrapper types into its corresponding primitive data type is known as **unboxing**.

Java I/O

- The differences between **BufferedReader** class and **Scanner** class are:

BufferedReader Class	Scanner Class
It's synchronous.	It isn't synchronous.

This class uses multiple threads.	This class uses a single thread.
It has an 8 KB byte buffer.	It has 1 KB char buffer.
It's a bit faster as compared to the Scanner class.	It's a bit slower as compared to BufferedReader class.

- ***Definition*:** Stream is defined as a sequence of data.
- In BufferedReader class:
  - The **read()** method is used for reading a **character** from the console in **ASCII** format.
  - The **readLine()** method is used for reading a **string** from the console.

Applet

- ***Definition*:** Applets are small Java applications that can be accessed on an internet server, transported over the internet, and can be automatically installed and run as part of a web document.
- The Applet class **doesn't** have any **main** method.
- Any applet in Java is a class that extends from the **java.applet.Applet** class.
- The life cycle of an applet is:
  - The **init()** method is used to initialise the applet.
  - The **start()** method is used to execute the applet.
  - The **paint()** method is used to draw the output of the applet in a separate window.
  - The **stop()** method is used to stop the execution of the applet.
  - The **destroy()** method is used to remove the applet from the memory.
- The **init()** and **destroy()** methods execute only **once**, while the other methods execute **multiple** times.
- The **start()** method executes **immediately after** the **init()** method is invoked.
- The **stop()** method executes **just before** the **destroy()** method is invoked.
- The **paint()** method is in **Graphics** class, whereas, all other methods are in **Applet** class.
- The method execution sequence when an applet is:

Executed	Closed
init() → start() → paint()	**stop() → destroy()**

- Applet HTML tag:

```
<applet code="Main" width="100px" height="100px"></applet>
```

- **code** attribute specifies the name of the applet.
- **width** attribute specifies the width of the applet in pixels.
- **height** attribute specifies the height of the applet in pixels.

AWT & Swing

- ***Definition*:** AWT is an **Abstract Window Toolkit** that provides various component classes like label, button, text field, etc. to show window components on the screen.
- ***Definition*:** Swing is a part of JFC (**Java Foundation Classes**), built on the top of AWT, and written entirely in Java.
- The differences between **AWT** and **Swing** are:

AWT	Swing
The AWT component classes are provided by the **java.awt** package.	The Swing component classes are provided by **javax.swing** package.
It's platform **dependent**.	It's platform **independent**.
It's **heavy** weight.	It's **light**weight.
Its appearance **depends** on the UI of the OS.	Its appearance **doesn't depend** on the UI of the OS.
It provides a **smaller** number of components.	It provides a **larger** number of components.
Needs a **large** amount of memory for execution.	Needs a **small** amount of memory for execution.
It's **slower** than Swing.	It's **faster** than AWT.

- The **java.awt** package contains the core **AWT graphics classes**.
- The **java.awt.event** package supports **event handling**.
- There are **two** types of GUI elements:

Component	Container
It represents an object with graphical representation.	It's a component that can contain other swing components.
Ex: Button, Label, Text Field, etc.	Ex: Frame & Panel.

- ***Definition*:** A frame is a top level container of an AWT program.

- ***Definition*:** A panel is a rectangular area used for grouping related GUI components in a certain layout.
- In a GUI program, a component must be kept in a container.
- Every UI has 3 main aspects:
    - UI elements.
    - Layouts.
    - Behaviours.
- **JComponent** is a **base class** for all swing UI components.